An improved batch processing CP approach

Sebastian Kosch

A thesis submitted in conformity with the requirements
for the degree of *Bachelor of Applied Science*

Division of Engineering Science
University of Toronto
2013

Contents

1 Introduction **1**

2 Fundamentals **3**

2.1 Linear programming models 3

2.1.1 Problem motivation 3

2.1.2 Graphical explanation 3

2.2 Integer programming models 3

2.2.1 Solution technique 3

2.2.2 Other methods 3

2.3 Constraint programming models 3

2.3.1 Concept 3

2.3.2 Solution 3

2.3.3 Global constraints 3

3 Modelling the problem **5**

3.1 Characteristics of the problem 5

3.2 Bounds on variables known a priori 5

3.2.1 Lower bound on $L_{\max}$ 5

3.2.2 Upper bound on $L_{\max}$ 6

3.2.3 Bounding the number of batches n^k 6

3.3 MIP model used 7

3.3.1 Original approach 7

3.3.2 Grouping empty batches 7

3.3.3 No postponing of jobs to later batches 8

3.4 CP model 9

3.4.1 Bin-packing and cumulative constraints 9
3.4.2 Temporal constraints 9
3.4.3 Constraint on number of batches with length $P_k > p$ 10
3.4.4 Temporal constraints on a job's start date 11
3.4.5 Restricting the search to ***before*** $L_{\max}$ 11
3.4.6 Grouping empty batches 12
3.4.7 No postponing of jobs to later batches 12
3.5 Decomposition approach 13
3.5.1 Branch-and-bound by batch in chronological order 13
3.5.2 Potential improvements 17
3.6 Open questions 17
3.7 Test data for evaluation 17

4 My solution **19**
4.1 MIP formulation improvements 19
4.2 CP formulation improvements 19

5 Discussion **21**
5.1 Intro 22
5.1.1 Chapter 8: Integer Programming 23
5.1.2 Chapter 9: Building IP models I. 24
5.1.3 Special kinds of IP models 25
5.1.4 How to formulate a good model 26
5.2 How this could be solved 27
5.2.1 Possible improvements 27

Introduction

Fundamentals

Some introductory paragraphs on minimization/maximization problems.

2.1 Linear programming models

An introduction to and illustration of LP models.

2.1.1 Problem motivation

2.1.2 Graphical explanation

2.2 Integer programming models

An introduction to and illustration of MIP models

2.2.1 Solution technique

2.2.2 Other methods

2.3 Constraint programming models

2.3.1 Concept

2.3.2 Solution

2.3.3 Global constraints

Modelling the problem

3.1 Characteristics of the problem

...

3.2 Bounds on variables known a priori

3.2.1 Lower bound on $L_{\max}$

A lower bound on $L_{\max}$ can be found using the lower bound on the completion date of each bucket q, where a bucket is defined as the set of batches with due date δ_q:

$$L_{\max} \geq C_{\max,q} - \delta_q \quad \forall q \tag{3.1}$$

This works because the buckets up to bucket q are guaranteed to contain all jobs with due dates $d \leq \delta_q$, since the batches within the buckets are ordered by earliest due date (EDD) in the optimal solution. The buckets up to bucket q will likely also contain some later ($d > \delta_q$) jobs in the optimal solution but this does not affect the validity of the lower bound.

Now we need to find $C_{\max,q}$, or at least a lower bound on it, in polynomial time. The simplest approach simply considers the "total elastic area", i.e. the sum of all $s_j p_j$ products:

$$C_{\max,q} \geq \lceil \frac{1}{b} \sum_{j:d_j \leq \delta_q} s_j p_j \rceil \quad \forall q \tag{3.2}$$

A better lower bound on $C_{\max,q}$ would be given by a preemptive-cumulative schedule. Unfortunately, minimizing $C_{\max}$ for such problems is equivalent to solving a standard bin-packing problem, which requires exponential time.

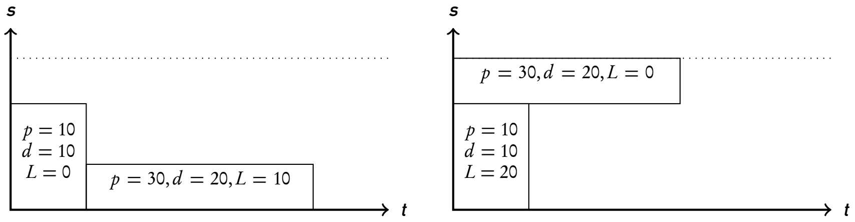


Figure 3.1: Overzealous batch elimination can increase L_{max}

3.2.2 Upper bound on L_{max}

An upper bound on L_{max} can be found by using a dispatch rule to find a feasible, if not optimal, schedule. A good approach could be the "best-fit" heuristic proposed in the original paper. This has not been implemented yet.

3.2.3 Bounding the number of batches n^k

Initially, the number of batches needed is assumed to be equal to the number of jobs: $n_k = n_j$. Reducing n_k by pre-computing the maximum number of batches needed shrinks the x_{jk} matrix.

Unfortunately, we cannot make a general statement that optimal solutions never have more batches than other feasible solutions – a simple counterexample is shown in figure 3.1.[1]

...

...

I have a long list of ideas here, none of which I've been able to prove right or wrong, so I've commented them out and made this page end here.

[1] To be more precise, we cannot state that at least one optimal solution is in the subset of feasible solutions that uses the fewest number of batches – a dominance situation that could be exploited, were it true.

3.3 MIP model used

3.3.1 Original approach

This is Malapert's original approach:

$$\sum_{k\in K} x_{jk} = 1 \qquad \forall j \in J \tag{3.3}$$

$$\sum_{j\in J} s_j x_{jk} \leq b \qquad \forall k \in K \tag{3.4}$$

$$p_j x_{jk} \leq P_k \qquad \forall j \in J, \forall k \in K \tag{3.5}$$

$$C_{k-1} + P_k = C_k \qquad \forall k \in K \tag{3.6}$$

$$(d_{max} - d_j)(1 - x_{jk}) + d_j \geq D_k \qquad \forall j \in J, \forall k \in K \tag{3.7}$$

$$D_{k-1} \leq D_k \qquad \forall k \in K \tag{3.8}$$

$$C_k - D_k \leq L_{\max} \qquad \forall k \in K \tag{3.9}$$

$$C_k \geq 0, P_k \geq 0 \text{ and } D_k \geq 0 \qquad \forall k \in K \tag{3.10}$$

The constraint 3.8 is the EDD constraint.

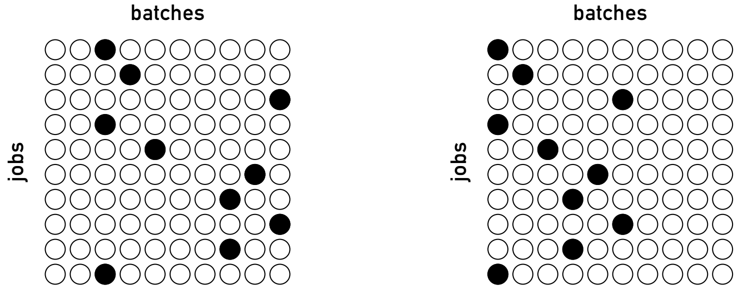
3.3.2 Grouping empty batches

The given formulation lacks a rule that ensures that no empty batch is followed by a non-empty batch. Empty batches have no processing time and a due date only bounded by $d_{\max}$, so they can be sequenced between non-empty batches without negatively affecting $L_{\max}$. Since, however, desirable schedules have no empty batches scattered throughout, we can easily reduce the search space by disallowing such arrangements. The idea is illustrated in Figure 3.2.

A mathematical formulation is

$$\sum_{j\in J} x_{j,k-1} = 0 \rightarrow \sum_{j\in J} x_{jk} = 0 \quad \forall k \in K. \tag{3.11}$$

To implement this, we can write constraints in terms of an additional set of binary variables, e_k, indicating whether a batch k is empty or not:



(a) Without dominance rule

(b) With dominance rule

Figure 3.2: Dominance rule to eliminate empty batches followed by non-empty batches

$$e_k + \sum_{j \in J} x_{jk} \geq 1 \qquad \forall k \in K, \tag{3.12}$$

$$n_j(e_k - 1) + \sum_{j \in J} x_{jk} \leq 0 \quad \forall k \in K. \tag{3.13}$$

Constraint 3.12 enforces $e_k = 1$ when the batch k is empty. Constraint 3.13 enforces $e_k = 0$ otherwise, since the sum term will never exceed n_j. The rule 3.11 can now be expressed as $e_{k-1} = 1 \rightarrow e_k = 1$, and implemented as follows:

$$e_k - e_{k-1} \geq 0 \quad \forall k \in K. \tag{3.14}$$

We can also prune any attempts to leave the first batch empty by adding a constraint $e_0 = 0$.

This, while it works just fine, actually takes longer than without it.

3.3.3 No postponing of jobs to later batches

Since the jobs are already sorted by non-decreasing due dates, it makes sense to explicitly instruct the solver never to attempt to push jobs into batches with a greater index than their own: even if every job had its own batch, it would be unreasonable to ever postpone a job to a later batch.

$$x_{jk} = 0 \quad \forall j,k : j > k \tag{3.15}$$

3.4 CP model

[Some introduction ...]

3.4.1 Bin-packing and cumulative constraints

This makes sure the jobs are distributed into the batches such that no batch exceeds the capacity b:

$$\texttt{pack}(J, K, b) \tag{3.16}$$

The cumulative constraint functions similarly, but instead of packing discrete bins, it enforces non-overlapping constraint on the temporal (`IntervalVariable`) J variables. For this to work directly with the `assignments` array used by `pack` (i.e. without a x_{jk} matrix), it seems I need to define my own search goal class – I still need to do this).

$$\texttt{cumul}(J, b) \tag{3.17}$$

3.4.2 Temporal constraints

These constraints are implemented using `IntervalVar` objects, which offer properties such as `lengthOf` or `endOf`.

$$P_k \geq \max_{j:B_j=k} p_j \qquad \forall k \in K \tag{3.18}$$

$$D_k \leq \min_{j:B_j=k} d_j \qquad \forall k \in K \tag{3.19}$$

$$C_k + P_{k+1} = C_{k+1} \qquad \forall k \in K \tag{3.20}$$

$$L_{\max} \geq \max_k (C_k - D_k) \tag{3.21}$$

The first constraint ensures that each batch is as long as its longest job. The second constraint ensures that the earliest-due job sets the due date. The third constraint defines batch completion dates, and the fourth constraint defines $L_{\max}$.

The first two temporal constraints were (rather, will be) implemented in custom constraint propagator classes using the job assignments propagated by the bin-packing constraint. They exploit the temporal features of `IntervalVars`.

3.4.3 Constraint on number of batches with length $P_k > p$

Since batches take on the processing time of their longest job, there is at least one batch with $P = \max_j p_j$:

$$\texttt{globalCardinality}(|P_k = \max_j p_j| = 1) \tag{3.22}$$

We can proceed to fill batches with jobs, ordered by non-increasing processing time, based on algorithm 1.

Algorithm 1 Generating lower bounds on batch lengths

```
$J^\star \leftarrow J$                                              ▷ initialize all jobs as unassigned jobs
$n_k \leftarrow 1$; $S_k \leftarrow \{0\}$; $P_{k,\min} \leftarrow \{0\}$          ▷ Create one empty batch of size and length zero
sort $J^\star$ by processing time, non-increasing
repeat
    $j \leftarrow J^\star$.pop()                                   ▷ select job for assignment, longest job first
    loop through all $n_k$ existing batches $k$, first batch first
        $k_p \leftarrow \varnothing$                                            ▷ no feasible batch
        $c_{\min} = b$                                  ▷ currently known minimum remaining capacity
        if $s_j < b - S_k$ and $b - S_k < c_{\min}$ then
            $k_p \leftarrow k_p$; $c_{\min} \leftarrow b - S_k$
        end if
    end loop
    if $|k_p| = 1$ then
        $S_{k_p} \leftarrow S_{k_p} + s_j$                                   ▷ assign job $j$ to batch $k_p$
    else
        if $n_k < LB(n_k)$ then
            $n_k \leftarrow n_k + 1$                                        ▷ open new batch
            $S_{n_k} \leftarrow s_j$; $P_{n_k,\min} \leftarrow p_j$                ▷ assign $s_j$ and $p_j$ to the new batch
        else
            leave the loop now and end.
        end if
    end if
until $J^\star$ is empty
```

At the end of this algorithm, we can state:

$$\texttt{globalCardinality}(|P_{k-1,\min} > P_k \geq P_{k,\min}| = 1) \quad \forall k \in \{k_0, \dots, k_{LB(n_k)}\} \tag{3.23}$$

The algorithm sorts jobs by non-increasing p, and then fills batches job by job. If a job fits into a previous batch, it is assigned there. If a job fits into multiple previous batches, it is assigned to the batch with the smallest remaining capacity. As I found out today, this is called *best-fit decreasing rule*. I can expand on why this works, if necessary.

Unfortunately, the optimal solution may perform better in terms of "vertical" (s_j) bin packing, and may thus require fewer batches. We therefore need to find a lower bound $LB(n_k)$ on the number of batches, and we can only guarantee the first $LB(n_k)$ of the above constraints to hold in the optimal solution. Finding a true lower bound is a two-dimensional bin packing problem, which runs in exponential time … so we have to come up with an even lower bound – right now I can only think of j_0, the number of jobs ordered by decreasing s_j that can never fit into a batch together.

This whole method can surely be improved somehow by realizing a priori that some jobs must go before others?

Furthermore, if all jobs have different processing times, all batches will have different processing times as well: `alldifferent`(P_k). If m out of n_j jobs have different processing times, we can still enforce `k_alldifferent`(P_k, m). Propagation rules for this constraint were given in [?]. I can't find anything on this w.r.t. CP Optimizer, so I may have to implement this myself … time permitting.

3.4.4 Temporal constraints on a job's start date

Given any partial assignment of jobs and an open job j, we can reason that

- if the first batch with a due date later than the job is k, then the job cannot be part of a batch after k – this would result in a non-EDD sequence of batches.
- if the first batches up to $k-1$ offer not enough capacity for j due to the given partial assignment, then the job cannot be part of a batch before k.

Since batches are *not* dynamically created like in Malapert's solution but fixed from the start, any partial assignment that fails due to these constraints cannot be part of an optimal solution.

This constraint is redundant with both the $(C_{k+1} \geq C_k)$ and `packing` constraints, but may help accelerate the propagation in some cases.

3.4.5 Restricting the search to *before* $L_{\max}$

Once a feasible (but possibly non-optimal) solution is found that has $L_{\max}$ in batch $k_{L_{\max}}$, then this solution dominates all solutions with other job configurations *after* $k_{L_{\max}}$ – improving L in those batches has no impact on $L_{\max}$, and worsening it never leads to a better solution.

Therefore, once all jobs are assigned to a feasible solution, the search should be limited to the following jobs:

- those jobs in the first batch that are due with the second batch k_2 or later,
- all jobs in batches $\{k_2, \ldots, k_{L_{\max}-1}\}$,
- all jobs in $k_{L_{\max}}$ for which $p_j > j_{L_{\max}}$,
- $j_{L_{\max}}$ itself, but it can only be scheduled earlier.

Once another, better feasible solution is found, this constraint on "searchable" variables must be updated. If no better solution is found using those variables, the solution is optimal. I think this could maybe speed up the search, but I have no idea how to implement this dominance rule as a constraint. The documentation offers `SearchPhase` functions that can guide the search, but I haven't experimented with that yet.

3.4.6 Grouping empty batches

Just like in the MIP model, we can force empty batches to the back and thus establish dominance of certain solutions. The implementation is much easier than in the MIP model:

$$\texttt{IfThen}(C_{k+2} > C_k, C_{k+1} > C_k) \quad \forall k \in \{k_1, \ldots, k_{n_k-2}\} \tag{3.24}$$

3.4.7 No postponing of jobs to later batches

Just like in the MIP model, jobs should never go into a batch with an index greater than their own:

$$x_{jk} = 0 \quad \forall j, k : j > k \tag{3.25}$$

This is implemented as $\texttt{assignments}_j \leq k$.

This constraint is analogous to the concept of finding an upper bound on n_k – essentially, it would be very helpful to find an upper bound on the latest batch for *every* job.

3.5 Decomposition approach

Instead of solving the entire problem using one model, we can solve the problem step by step, using the best techiques available for each subproblem. This approach is inspired by a method called *Benders' decomposition*.

3.5.1 Branch-and-bound by batch in chronological order

A basic version of this approach uses branch-and-bound to transverse the search tree. At each node on level ℓ, a single MIP and/or CP model is run to assign jobs to batch $k = \ell$. The remaining jobs are passed to the children nodes, which assign jobs to the next batch, and so on – until a solution, and thus a new upper bound on $L_{\max}$, is found. Several constraints are used to prune parts of the search tree that are known to offer only solutions worse than this upper bound. Figure 3.3 shows an example in which a MIP model is used to assign jobs to the batch at every node. Algorithm 2 outlines what happens at each node: the model finds the best jobs to assign to the batch according to some rule, lets the children handle the remaining jobs, and tries the next best solution once the first child has explored its subtree and backtracked.

Algorithm 2 MIP node class code overview

```
update currentAssignments                              ▷ this keeps track of where we are in the tree
if no jobs given to this node then                     ▷ if this is a leaf node, i.e. all jobs are assigned to batches
    calculate $L_{\max,\text{current}}$ based on currentAssignments
    if $L_{\max,\text{current}} < L_{\max,\text{incumbent}}$ then
        $L_{\max,\text{incumbent}} \leftarrow L_{\max,\text{current}}$
        bestAssignments ← currentAssignments
    end if
    return to parent node
end if
set up MIP model                                       ▷ as described below
repeat
    $x_j \leftarrow$ model.solve($x_j$)                ▷ let model assign jobs to the batch
    spawn and run child node with all $\{j | x_j = 0\}$    ▷ pass unassigned jobs to children
    add constraint to keep this solution from recurring    ▷ this happens once the child node returns
until model has no more solutions
return to parent node
```

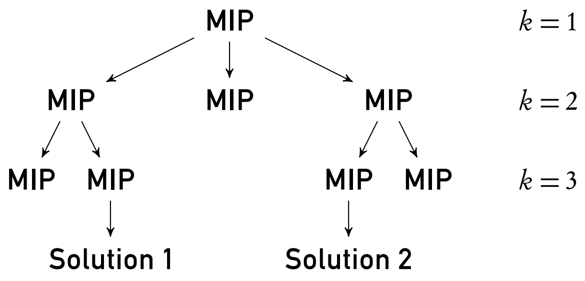


Figure 3.3: Batch-by-batch decomposition using MIP only

Using MIP and cumulative packing after the batch

The first version of the branch-and-bound batch-by-batch decomposition uses a MIP model at each node to assign jobs to the respective batch. The remaining jobs are packed such as to minimize their $L_{\max}$, with a relaxation of the batching requirement, i.e., as if on a cumulative resource. This is implemented using cumulative constraints over indexed time points as shown in Model 3.5.1.

x_j	is 1 iff job j is assigned to the batch
u_{jt}	is 1 iff job j starts in time slot t
T_{jt}	is the set of all time slots occupied by job j if it ended at time t, that is $T_{jt} = \{t - p_j + 1, \ldots, t\}$
v_k	is the start time of the batch at the given node in the search tree
$L_{\max,\text{incmb}}$	is the incumbent (known best) value of and thus an upper bound on $L_{\max}$
$\mathscr{H}$	is the set of all indexed time points $\{1, \ldots, n_t\}$
w_j	is 1 iff job j is either longer than the batch ($p_j > P_k$) or already part of the batch ($x_j = 1$)

Table 3.1: Notation used in the decomposition model

The above model implements a time-indexed cumulative constraint on the non-batched jobs. Constraints (3.27) through (3.29) ensure that the batch stays below capacity, define the duration of the batch P_k and force at least one of the earliest-due jobs into the batch.

Constraints (3.30) express the interval relaxation used to ensure that no jobs exceed their latest allowable finish date given any batch assignment. Even jobs that are assigned to the batch have to fulfill this requirement.

$$\text{Min.} \quad L_{\text{max,cumul}} \tag{3.26}$$

$$\text{s. t.} \quad \sum_{j} s_j x_j \leq b \qquad \forall j \in J \tag{3.27}$$

$$P_k \geq p_j x_j \qquad \forall j \in J \tag{3.28}$$

$$\sum_{j} x_j \geq 1 \qquad \forall \{ j \in J | d_j = \min(d_j) \} \tag{3.29}$$

$$P_k + \frac{1}{b} \sum_{i} s_i p_i \leq d_j + L_{\text{max,incmb}} - 1 - v_k \qquad \forall j \in J, \forall \{ i \in J | d_i \leq d_j \} \tag{3.30}$$

$$\sum_{t} u_{jt} = 1 \qquad \forall j \in J \tag{3.31}$$

$$\sum_{j} \sum_{t' \in T_{jt}} u_{jt'} \leq b \qquad \forall t \in \mathscr{H} \tag{3.32}$$

$$(v_k + t + p_j) u_{jt} \leq d_j + L_{\text{max,incmb}} - 1 \qquad \forall j \in J, \forall t \in \mathscr{H} \tag{3.33}$$

$$L_{\text{max,cumul}} \geq (v_k + t + p_j) u_{jt} - d_j \qquad \forall j \in J, \forall t \in \mathscr{H} \tag{3.34}$$

$$u_{j,t=1} = x_j \qquad \forall j \in J \tag{3.35}$$

$$u_{it} \leq (1 - x_j) \qquad \forall i, j \in J, \forall t \in \mathscr{H} \setminus \{2\} \tag{3.36}$$

$$b - \sum_{j} s_j x_j \leq (b w_j + 1) s_j \qquad \forall j \in J \tag{3.37}$$

$$P_k + 2 w_j n_t \geq p_j + n_t x_j \qquad \forall j \in J \tag{3.38}$$

$$P_k - 2(1 - w_j) n_t \leq p_j + n_t x_j - 1 \qquad \forall j \in J \tag{3.39}$$

Model 3.5.1: MIP model in batch-by-batch branch-and-bound

Constraints (3.31) and (3.32) implement the cumulative nature of the post-batch assignments by ensuring that each job starts only once, and no jobs overlap on a given resource at any time (for the purposes of this model, the batch machine is considered divisible into b unary resources).

Constraints (3.33) again limit the possible end dates of a job, but unlike (3.30), they use the time assignments on the cumulative resource to determine end dates. Constraints (3.34) define the value of $L_{\text{max,cumul}}$, the maximum lateness of any job in the non-batched set.

Constraint (3.35) force batched jobs to start at $t = 0$, while (3.36) force *all* jobs to start either at $t = 0$ or after the last batched job ends.

Constraints (3.36) enforce a dominance rule: jobs must be assigned to the batch such that the remaining capacity, $b_r = b - \sum_j s_j x_j$, is less than the size s_j of the *smallest* job from the set of non-batched jobs that are *not longer* than the current batch. That is, if there exists an non-batched job j with $p_j \leq P_k$ and $s_j \leq b_r$, then the current assignment of jobs is infeasible in the model. The reasoning goes as follows: given any feasible schedule, assume there is a batch k_a with b_r remaining capacity and a later batch k_b containing a job j such that $s_j \leq b_r$ and $p_j \leq P_{k_a}$. Then job j can always be moved to batch k_a without negatively affecting the quality of the solution: if the schedule was optimal, then moving j will not affect $L_{\max}$ at all (otherwise, it was no optimal schedule); if the schedule was not optimal, then $L_{\max}$ will stay constant (unless j was the longest job in k_b and $L_{\max}$ occured in or in a batch after k_b, in which case $L_{\max}$ will be improved).

This rule is implemented by means of a binary variable w_j, which, as defined by constraints (3.37) and (3.38), is 1 iff $p_j > P_k \vee x_j = 1$. These are the cases in which a job j is *not* to be considered in (3.36), and so w_j is used to scale s_j to a value insignificantly large in the eyes of the constraint's less-than relation.

Using CP and cumulative packing after the batch

This approach is equivalent, but we now use CP to select the batch assignments, again based on a minimized $L_{\max}$ among the non-batched jobs.

3.5.2 Potential improvements

Improve the initial $L_{\max,\text{incumbent}}$ A better initial upper bound on $L_{\max}$ can help prune some branches of the search tree from the outset. There are several dispatch rules (or maybe other heuristics?) that could be explored to do this better.

Improve $L_{\max,\text{incumbent}}$ during search It may be useful to use a heuristic like above to "complete the schedule" once a promising partial schedule has been generated. I have yet to identify situations where this is always helpful.

3.6 Open questions

- Is there a dominance rule that reduces the size of the problem because two jobs can be "fused together" (WHATS THE WORD?) to be considered as one?

3.7 Test data for evaluation

Malapert uses benchmark data by Daste. For design purposes, I created my own set of randomized job lists, with $s_j, p_j \in [1, 20]$ and $d_j \in [1, 10n]$ where n is the number of jobs. Here is the Python code used:

```
import csv
import random

random.seed(None)

for i in range(10,100,20):
    with open( ”data_” + str(i), ”wb”) as csvfile:
        csvwriter = csv.writer(csvfile, delimiter=”,”)
        csvwriter.writerow([’s_j’, ’p_j’, ’d_j’])
        for job in range(i):
            csvwriter.writerow([random.randrange(1,20),
               random.randrange(1,20),
               random.randrange(1,5*i)])
    csvfile.close()
```

CSV files can be read in with `IloCsvReader` and `IloCsvLine`.

My solution

4.1 MIP formulation improvements

4.2 CP formulation improvements

Discussion

5.1 Introduction

Books to read: Model Building in Mathematical Programming by HP Williams

Mathematical programming models all involve optimization. They want to minimize or maximize some objective function.

There are linear programming models, non-linear programming models and integer programming models.

Constraint programming and Integer Programming are twins and can usually be translated into one another. In CP each variable has a finite domain of possible values. Constraints connect/restrict the possible combinations of values which the variables cantake. These constraints are of a richer variety then the linear constraints of IP and are usually expressed in the form of predicates, such as "all_different(x1, x2, x3)". The same condition could be imposed via IP, but it's more troublesome to formulate. Once one of the variables has been set to one of the values in its domain (either temporarily or permanently), this predicate would imply that this value must be taken out of the domain of all other variables ("propagation"). In this way constraint propagation is used until a feasible set of values is found (or not).

CP is useful where a problem function has no objective function and we are just looking for a feasible solution. We can't prove optimality, although we could using IP.

Comparisons and connectionsbetween IP and CLP are discussed by Barth (1995), Bockmayr and Kasper (1998), Brailsford et al. (1996), Prolland Smith (1998), Darby-Dowman and Little (1998), Hooker (1998) and Wilson and Williams (1998). Theformulation of the all_different predicate using IP is discussed by Williams and Yan (1999).

The first approach to solving a multi-objective problem is to solve the model a number of times with each objective function in turn. The result may give an idea of what to do next.

5.1.1 Chapter 8: Integer Programming

The real power of IP as a method of modelling is that of binary constraints, where variables can take on values of 0 or 1. Maybe it should be called "discrete programming." Precise definitions of those problems that can be formulated by IP models are given by Meyer (1975) and Jeroslow (1987).

Problems with discrete inputs and outputs are the most obvious candidates for IP modelling ("lumpy inputs and outputs"). Sometimes solving an LP and rounding to the nearest integer works well, but sometimes it doesn't, as demonstrated in Williams, first example in 8.2. The smaller the variables (say, <5), the greater significance those rounding problems will have.

When input variables have small domains, say, machine capacities, then again this may rule out LP relaxations. A good example of an IP problem is the knapsack assignment problem. A single constraint is that the capacity of the knapsack cannot be exceeded. Any LP formulation would always fill the knapsack 100%, ignoring the discrete size of the objects. Two other well-known types of problems include *set partitioning problems* and *aircrew scheduling problems*.

Integer programming models cannot be solved directly, but need to be brute-forced in a tree search manner. The search space, however, can be greatly reduced by a number of bounding techniques often depending on the nature of the problem. That is why the general method of IP solving is called *branch-and-bound*.

So-called *cutting planes methods* usually start by solving an IP problem as LP. If the resulting solution is integral, we're happy. Otherwise extra constraints (cutting planes) are added to the problem, further constraining it until an integer solution is found (or, if none can be found, we're out of luck). Cutting planes make for nice illustrations they are not very efficient with large problems. Cutting planes were invented by Gomory (1958).

Enumerative methods are generally applied to pure binary problems, where the search tree is pruned. The best known of these methods is Balas's additive algorithm described by Balas (1965). Other methods are givenby Geoffrion (1969). A good overall exposition is given in Chapter 4 of Garfinkel and Nemhauser (1972).

There are so-called *pseudo-boolean methods* to solve pure binary problems that take boolean constraints as inputs. That may be comfortable for the user sometimes, but is rarely used in practice.

Generally, branch-and-bound methods first solve the LP relaxation to check whether we're lucky enough to find an integer solution. If not, a tree search is performed.

5.1.2 Chapter 9: Building IP models I.

Binary variables are often called 01-variables. Decision variables could also have a domain like $\{0,1,2\}$ or $\{4,12,102\}$. Decision variables, especially the 01 kind, can be linked to the state of continuous variables like this: $x - M\delta \leq 0 \leftrightarrow x > 0 \rightarrow \delta = 1$, where we know that $x < M$ is always true.

To use a 01 variable to indicate whether the following is satisfied:

$$2x_1 + 3x_2 \leq 1 \tag{5.1}$$
$$x_1 \leq 1 \tag{5.2}$$
$$x_2 \leq 1, \tag{5.3}$$

so, mathematically speaking,

$$\delta = 1 \rightarrow 2x_1 + 3x_2 \leq 1 \tag{5.4}$$
$$2x_1 + 3x_2 \leq 1 \rightarrow \delta = 1, \tag{5.5}$$

Then we can argue that at most, $2x_1 + 3x_2 = 5$, so

$$2x_1 + 3x_2 + 4\delta \leq 5$$

will ensure that $\delta = 1$ forces the equation to be true: use $M = 2 + 3 - 1$ to find 4. In order to enforce $\delta = 1$ if the equation is true, use $m = 0 + 0 - 1$ and write

$$2x_1 + 3x_2 + \delta \geq 1$$

All kinds of logical conditions can be modelled using 01 variables, although it's not always obvious how to capture them in that format.

Logical conditions are sometimes expressed within a Constraint Logic Programming language as discussed inSection 2.4. The tightest way of expressing certain examples using linear programming constraints is described byHooker and Yan (1999) and Williams and Yan (1999). There is a close relationship between logic and 01 integerprogramming, which is explained in Williams (1995), Williams and Brailsford (1999) and Chandru and Hooker(1999).

Special Ordered Sets Two very common types of restriction arise in mathematical programming, so two concepts (SOS1 and SOS2) have been developed. An SOS1 is a set of variables within which exactly one variable must be non-zero and the rest zero. An SOS2 is a set where at most two can be non-zero, and the two variables must be adjacent in the input ordering. Using a branch-and-bound algorithm specialized for SOS1 or SOS2 sets can speed things up greatly.

Disjunctive constraints It is possible to define a set of constraints and postulate that at least one of them be satisfied.

5.1.3 Special kinds of IP models

Most practical IP models do not all into any of these categories but arise as MIP models often extending an existing LP model. Here are some examples.

Set covering problems We have a set $S = \{1, 2, 3, \dots, m\}$. We have a bunch of subsets $\mathscr{S}$ of subsets, each associated with a cost. Now cover all members of S using the least-cost members of $\mathscr{S}$.

Knapsack problem These are the really simple ones with just one constraint, namely the constraint of not being able to take more items with you than the knapsack can carry while maximizing the value of the taken objects.

Quadratic assignment problem Two sets of objects S and T of the same size require the objects to be matched pairwise. There are costs associated with pairs of pairs, that is the cost $c_{ij,kl}$ is the cost of assigning i to j while also assigning k to l. This cost will be incurred if both δ_{ij} and δ_{kl} are true, i.e. $\delta_{ij}\delta_{kl} = 1$. The objective function is a quadratic expression in 01 variables:

$$\text{Minimize} \sum_{\substack{i,j,k,l=1 \\ k>l}}^{n} c_{ij,kl}\delta_{ij}\delta_{kl} \tag{5.6}$$

The quadratic version is practically unsolveable, so to be able to enumerate the possible assignments the above objective function has to be split up into separate constraints, which obviously means there will be an explosion in problem size as the number of variables grow.

5.1.4 How to formulate a good model

According to Williams, it is easy to build IP models that, while correct, are really inefficient. Fortunately, with some knowledge of what happens behind the scenes (and some practice), sucky models can often be improved. One good method to know (although it doesn't help by itself) is to turn a general integer variable into a bunch of 01 variables. Say γ is a general, non-negative integer variable with a known upper bound of u, then we can replace it with $\delta_0 + 2\delta_1 + 4\delta_2 + 8\delta_3 + \cdots + 2^n\delta_n$.

Example problem http://www.scribd.com/doc/49547850/Model-Building-in-Mathematical-Programming

5.2 How this could be solved

We're trying to create an mixed integer linear programming model (MILP). In the original paper, they had one non-linear constraint:

$$(d_{max} - d_j)(1 - x_{jk}) \geq D_k - d_j \quad \forall j \in J, \forall k \in K \tag{5.7}$$

This can easily be turned into the simpler

$$D_k - (d_j - d_{max})x_{jk} \leq d_{max} \quad \forall j \in J, \forall k \in K \tag{5.8}$$

5.2.1 Possible improvements

We cannot modify the solution process itself – that's the whole point of the exercise, after all. Maybe we can preprocess to limit the domains of some decision variables. This is where "preordering" may come into play.

Limiting batch due date Since we start with as many batches as jobs, the solver is free to put every job into a batch of the same number. To make things more efficient, it can also put jobs into batches before, but it would never make sense to put a job into a batch with a higher number, so we have a limit that

$$x_{jk} = 0 \quad \forall j, k : j > k \tag{5.9}$$